# Versat-AI: An ONNX-to-SoC Compiler for Model-Agnostic CGRA Edge Inference

Rúben Teixeira , João Barreiros Rodrigues * , Jaime Aguiar , Jiao Li , José T. de Sousa *

Posted Date: 20 July 2026

doi: 10.20944/preprints202607.1442.v1

Keywords: edge inference; CGRA; ONNX; RISC-V SoC; FPGA; hardware accelerator; deep learning; hardware/software co-generation; neural network; open-source; low power; energy efficiency

 doi:10.20944/preprints202607.1442.v1



*Article*

# Versat-AI: An ONNX-to-SoC Compiler for Model-Agnostic CGRA Edge Inference

**Rúben Teixeira [1], João Barreiros Rodrigues [2,*], Jaime Aguiar [1,2], Jiao Li [3] and José T. de Sousa [1,2,*]**

[1] IObundle, Lda, Lisbon, Portugal
[2] INESC-ID; INESC INOV; Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal
[3] Microelectronics Research Development Center, School of Mechatronic Engineering and Automation, Shanghai University, Shanghai, 200444, China
* Correspondence: joao.b.rodrigues@inov.pt (J.B.R.); jose.t.de.sousa@tecnico.ulisboa.pt (J.T.d.S.)

**Abstract**

Edge inference on resource-constrained embedded nodes demands accelerators that are energy-efficient and compact. This paper presents Versat-AI, an open-source compiler that accepts a standard Open Neural Network Exchange (ONNX) model and generates a complete, synthesisable RISC-V System-on-Chip (SoC) with an embedded CGRA accelerator. The key innovation is applying a known hardware merge strategy to collapse structurally compatible neural network operators into a physical CGRA instance. The Versat-AI compiler also derives memory-mapped interconnects, firmware drivers, and RISC-V application software co-generated by the Py2HWSW SoC framework, eliminating the manual hardware/software co-design effort that previously tied this accelerator's own design lineage to a single target network. The next phase of the project is to extend this same automatic derivation from sizing the operator vocabulary to sizing per-operator parallel instancing and bandwidth, following the bandwidth-matched scaling principle already demonstrated, by hand, in this accelerator's own design lineage. The current phase of the project has succeeded in creating an sound automation flow that produces an accelerator that maps each operator onto a single physical datapath instance and occupies 8,763 LUTs, 9,833 flip-flops, 4 DSPs, and 202 BRAMs on a Xilinx Kintex Ultrascale field-programmable gate array (FPGA)—a footprint unchanged across all evaluated models regardless of size—and draws 0.65 W (1.96 W for the complete SoC including DDR4 controller), achieving 2.3× to 9× speedup over a software-only baseline on four MLPerf Tiny benchmark tasks. The paper also surveys the two-decade lineage of reconfigurable accelerators that motivated Versat-AI's design, and documents the open-source SoC platform development—including two prior integration attempts and the decisions that led to IOb-SoC.

**Keywords:** edge inference; CGRA; ONNX; RISC-V SoC; FPGA; hardware accelerator; deep learning; hardware/software co-generation; neural network; open-source; low power; energy efficiency

## 1. Introduction

The proliferation of deep learning has created demand for inference at the edge: smart sensors, industrial monitors, and personal devices must classify, detect, or predict in real time within strict power and area budgets. Fixed-function Application Specific Integrated Circuits (ASICs) satisfy those budgets but cannot adapt when models evolve; cloud offload reintroduces latency and connectivity dependence; Graphics Processing Units (GPUs) and high-end processors offer flexibility at power levels embedded nodes cannot sustain [1]. Coarse-Grain Reconfigurable Arrays (CGRAs) occupy the promising middle ground: word-level reconfigurability keeps area and energy close to ASIC levels while preserving post-fabrication programmability [2].

Three problems, however, have limited CGRA adoption in edge Artificial Intelligence (AI) deployment. First, the *compilation problem*: identifying and mapping kernels onto CGRA fabric—effectively High-Level Synthesis (HLS) for CGRAs—has resisted full automation for two decades; existing tools

 doi:10.20944/preprints202607.1442.v1



require expert intervention, do not accept high-level model descriptions as input, or both. Second, the *area scaling problem*: many CGRA-based accelerator designs sidestep temporal reuse by replicating a physical structure more than necessary, leaving the resulting design unsuitable for resource-constrained targets where silicon area is a hard constraint.

A complementary obstacle is the *SoC integration problem*: deploying any accelerator in a production-quality embedded system requires not only the accelerator Register Transfer Level (RTL) but also memory interfaces, bus wrappers, peripheral drivers, and firmware. Assembling such a system using standard tools has historically proved far harder than designing the accelerator itself, and has discouraged the use of CGRAs in production systems.

Versat-AI addresses all three challenges. It accepts a standard Open Neural Network Exchange (ONNX) [3] model as input and automatically produces a complete, synthesisable RISC-V System-on-Chip (SoC) with an embedded CGRA accelerator. The accelerator is generated by applying the hardware merge strategy of [4] to ONNX operator mapping: structurally compatible operators are collapsed into a CGRA instance, so the accelerator's hardware directly reflects the operator vocabulary the input model actually uses—rather than a designer-fixed, model-independent template or a hardware pipeline allocated per layer—keeping footprint nearly independent of model size. System integration is handled by the Py2HWSW [5] SoC generator, which produces memory-mapped interconnects, Control and Status Register (CSR) drivers, and firmware from the same description. The contributions of this work are:

- **Hardware merge strategy for ONNX operators.** The unit merge and reuse strategy of [4]—previously applied to DSP coprocessors—is applied here for the first time to neural network operator mapping. Structurally compatible operators are collapsed into CGRA configurations, producing an accelerator whose area is nearly independent of model size.
- **Automated Model-to-Runtime flow.** Versat-AI co-generates the hardware accelerator, memory-mapped interconnects, firmware drivers, and Reduced Instruction Set Computer V (RISC-V) application software from a single ONNX input, integrating with the Py2HWSW [5] SoC generator for a fully automated path from model to running silicon.
- **MLPerf Tiny evaluation.** The system is validated on four MLPerf Tiny inference tasks [6], achieving 2.3× to 9× speedup over a software-only baseline on a Xilinx Kintex Ultrascale Field Programmable Gate Array (FPGA) with a single automatically-derived datapath instance per operator—evidence that the automation itself is sound, ahead of extending it to automatically size per-operator parallel instancing and bandwidth.
- **Architectural lineage and SoC integration context.** The background section traces the two-decade evolution of reconfigurable accelerators—from SideWorks and Versat through DeepVersat and the YOLO accelerator—that shaped Versat-AI's design. It also documents the open-source SoC platform development, including two prior integration attempts with OpenRISC and the Rocket Chip generator, and the design decisions that led to IOb-SoC.

## 2. Related Work

### *2.1. CGRAs and Programmable Accelerators*

Coarse-grained reconfigurable arrays sit between FPGAs and fixed-function ASICs in the programmability–efficiency space: word-level reconfigurability keeps area and power close to ASIC levels while preserving post-fabrication flexibility [2]. Early architectures such as RaPiD [7] and ADRES [8] established the canonical template of interconnected processing elements with local memories, but both relied on offline loop-kernel mapping tools that never achieved the automation level of FPGA place-and-route, limiting practical adoption. That tooling gap has driven sustained research effort: Morpher [9] integrates CGRA design, compilation, and simulation into a single open-source framework targeting edge AI kernels, while RipTide [10] and Snafu [11] compile arbitrary C programs into energy-minimal spatial dataflow graphs, demonstrating that near-ASIC efficiency is achievable with the right compiler substrate. OpenEdgeCGRA [12] contributes an open-hardware CGRA specifi-

 doi:10.20944/preprints202607.1442.v1



cally evaluated on convolutional layers at the edge, providing a useful area and throughput baseline for CNN-targeted designs.

Despite this progress, all of these approaches require an explicit kernel mapping step—either offline scheduling or C-level entry—and none provides a direct path from an ONNX model to a synthesisable CGRA SoC: even where a compiler front-end exists, its output is the CGRA accelerator or its configuration alone, not the surrounding interconnects, firmware, and application software a deployable SoC requires. Versat addresses the mapping difficulty through self-reconfiguration: the RISC-V host programs the array at runtime from software, replacing the offline spatial mapper with a conventional instruction stream. The compiler layer above it—Versat-AI—further applies the hardware merge strategy of [4] to collapse structurally compatible ONNX operators into CGRA configurations.

### *2.2. Edge AI SoC Platforms*

The dominant open-source approach to edge AI SoC design is the Parallel Ultra-Low-Power (PULP) platform [13], whose commercial derivative GAP8 pairs a single-core controller domain with a cluster of eight RISC-V cores and achieves competitive energy efficiency on quantised inference workloads. The PULP-NN library [14] provides hand-tuned quantised inference kernels for this multi-core substrate, covering common Convolutional Neural Network (CNN) and Digital Signal Processor (DSP) primitives. The successor GAP9 extends the cluster to nine cores, adds floating-point support across all cores, and targets 50 GOPS at under 50 mW on a 22 nm process—making it the principal industrial benchmark for ultra-low-power edge inference. These platforms demonstrate that a tightly-coupled RISC-V cluster paired with a shared Tightly Coupled Data Memory (TCDM) memory is highly effective for data-parallel workloads, but inference relies on manually written, architecture-specific libraries rather than an automated model-to-hardware flow.

On the open-source SoC platform side, X-HEEP [15] provides a configurable and fully open-source RISC-V microcontroller with a standardised accelerator interface (XAIF) that has been taped out in 65 nm CMOS, making it the closest architectural cousin to IOb-SoC in terms of openness and configurability, though it does not itself generate an accelerator from a model. Gemmini [16], integrated into the Chipyard [17] SoC generator ecosystem, attaches a parametric systolic array to a Rocket RISC-V core via the RoCC interface and provides a full quantised execution flow from ONNX, but the array's dimensions are a design-time choice fixed independently of any particular model. CGRA4ML [18] goes further still, producing a complete, taped-out SoC—RTL, AXI Direct Memory Access (DMA) interfaces, firmware, and Arm integration— built around a CGRA that shares resources across network layers through runtime-multiplexed processing elements; however, its array dimensions are likewise a fixed, user-specified template, and its front end accepts only QKeras-trained models rather than a standard interchange format. None of these three, unlike Versat-AI, derives the accelerator's hardware configuration automatically from the operator vocabulary of an arbitrary ONNX model.

### *2.3. Inference Compiler Frameworks*

At the software compilation end, Apache TVM [19] and ONNX Runtime lower ONNX graphs into optimised kernels for existing processors and accelerators without generating hardware. FINN [20] and hls4ml [21] take the next step by synthesising model-specific hardware: FINN produces streaming pipelines of processing elements matched to the network topology for FPGAs, while hls4ml generates HLS blocks for low-latency scientific applications. Both are mature, widely used frameworks, but the hardware they produce scales in resource usage with model depth and width, making them unsuitable for area-constrained targets where the FPGA fabric is a hard limit. CONVOLVE [22] addresses CGRA targets specifically, lowering ONNX through an MLIR pipeline to an OpenASIP-based CGRA back-end with automated kernel scheduling. Vitis AI [23] provides a complete deployment flow for AMD FPGA devices using the Deep-learning Processing Unit (DPU) overlay, but targets mid-to-high-density FPGAs and is not open-source below the library level.

Versat-AI occupies a distinct position in this landscape: it applies the hardware merge strategy of [4] to ONNX operator mapping, identifying structurally compatible operators across different graph

 doi:10.20944/preprints202607.1442.v1



nodes and collapsing them into CGRA configurations. The result is a fixed-footprint accelerator whose area is independent of model size, integrated with the Py2HWSW [5] SoC generator to produce a complete synthesisable RISC-V SoC—including memory-mapped interconnects, peripheral wrappers, and firmware drivers—from a single ONNX input.

## 3. Background

### *3.1. Reconfigurable Accelerator Architecture*

Versat-AI draws its core architectural ideas from a line of reconfigurable accelerators developed over two decades, each resolving a limitation of its predecessor. The thread runs from SideWorks (hardware merge for DSP coprocessors) through Versat (generalised mini-CGRA with full crossbar) and DeepVersat (pipelined deep-learning array) to the YOLO accelerator, where the bandwidth-disciplined design philosophy and the VRead/VWrite memory units that Versat-AI inherits directly were first introduced.

#### 3.1.1. SideWorks and Versat

Coreworks SA, founded in 2001 from INESC-ID and Instituto Superior Técnico, needed to support a growing portfolio of DSP codecs—MP3, AAC, HE-AAC, Dolby Digital—without multiplying silicon area with each new algorithm. The answer was SideWorks [24]: a reconfigurable coprocessor in which structurally compatible operators from different algorithm modules are merged into a single shared physical instance via a partial read/write crossbar, so that silicon footprint is determined by the operator vocabulary rather than the number of supported algorithms [4]. The key mechanisms are a configurable multi-level Address Generation Unit (AGU) that reproduces any stride-based access pattern through reconfiguration alone, a programmable timing unit that propagates enable signals alongside data to compensate for unequal pipeline latencies automatically, and fine-grained partial reconfiguration at individual register granularity so that switching operators requires only a small number of CSR writes. BDTI independently validated the approach in 2009: the SideWorks-based CWcomp4465 core achieved a BDTIsimMark2000 score of 2440 in only 0.7 $mm^2$ at 49 mW (130 nm)—the smallest footprint in the benchmark comparison by more than a factor of two [25]. This hardware merge strategy is the direct conceptual precursor to Versat-AI, where it is applied for the first time to ONNX neural network operators.

Versat [26,27] generalised the merge concept into a domain-independent mini-CGRA by replacing SideWorks's fixed operator vocabulary with a implicit, full crossbar interconnect among 15 heterogeneous functional units as illustrated in Figure 1, eliminating the placement-and-routing step that dominates mapping tool complexity in conventional 2D-mesh CGRAs [2] at only 4% area overhead. This full-crossbar generality comes at a routing cost that grows quadratically with the number of functional units: Versat's footprint is fixed for its 15-unit vocabulary, but the same approach does not scale to a larger functional-unit set without the interconnect itself becoming the dominant, unroutable cost. A 16-instruction controller generates configuration sequences at runtime from compact parametric routines (*self-reconfiguration*), with a shadow register hiding reconfiguration latency behind the current execution phase. On 130 nm, Versat occupies 5.2 $mm^2$—smaller than ADRES [8] by 1.62× and with 2.66× lower energy than MorphoSys [28]—and delivers a 25.6× speedup and 311× energy reduction on an FFT kernel relative to an ARM Cortex-A9 [27]. DeepVersat [29] subsequently extended Versat into a linear pipeline of identical stages to meet the compute density of deep convolutional networks, mapping one network layer per hardware stage and forwarding intermediate feature maps directly between stages without external memory round-trips. Chaining stages this way, rather than enlarging Versat's crossbar, was the point: it scales usable compute capacity through local, stage-to-stage connections instead of a single interconnect whose routing cost would explode with functional-unit count. The approach proved effective, but its reuse is by *repetition*: each stage is a full, homogeneous copy of the same elemental Versat block, and the pipeline's depth and topology are fixed at synthesis time to one specific network—a different model requires instantiating a new pipeline, not because area is

 doi:10.20944/preprints202607.1442.v1



forced to grow with depth, but because the design reuses identical stages rather than sharing them. These limitations directly motivated the compiler-driven merge strategy at the core of Versat-AI, which instead aggregates *heterogeneous* functional units into a single shared structure and reuses the routing itself across many different operator configurations—even when the units each configuration exercises differ—keeping interconnect limited to what is strictly necessary rather than a general crossbar or a chain of identical copies.

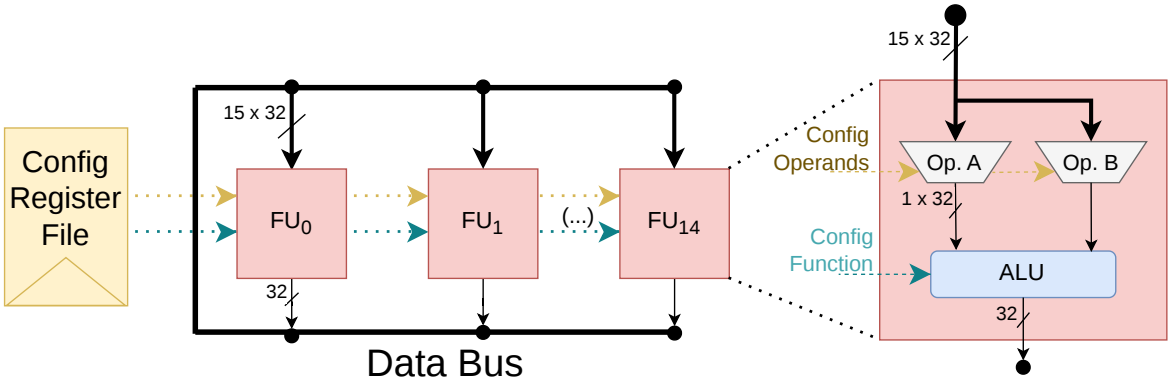


**Figure 1.** Original Versat's Data Engine structure.

3.1.2. Versat for Object Detection

Rather than extending DeepVersat, a different path was taken in [30,31]: a hand-crafted accelerator designed from first principles around the bandwidth constraints of Tiny-YOLO inference, integrated into a complete RISC-V SoC on a Kintex Ultrascale KU040 FPGA. A topview of the YOLO Versat architecture is illustrated in Figure 2. The work established two principles and one hardware abstraction that Versat-AI inherits directly.

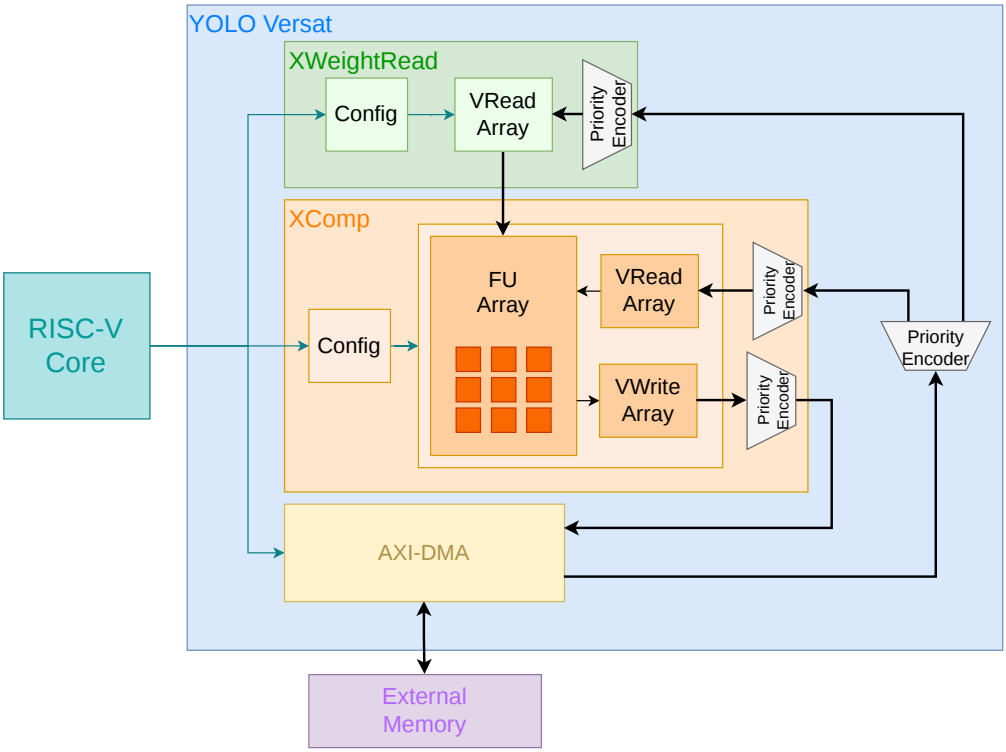


**Figure 2.** YOLO Versat Architecture topview.

The first principle is that memory bandwidth, not arithmetic throughput, is the binding constraint in CNN inference [1]: adding Multiply And Accumulate (MAC) units beyond the point the memory bus can sustain yields no throughput gain. The design therefore sizes compute to match the available

 doi:10.20944/preprints202607.1442.v1



bus, resulting in 832 (YOLOv3-Tiny) or 1,248 (YOLOv4-Tiny) MAC units fed by a 256-bit DMA interface, and ensures that the bus is kept fully utilised at all times. Versat-AI carries this discipline forward: the current implementation instantiates ONNX operators of different types and makes sure the units are never data-starved.

The second principle is full pipeline coverage. Unlike prior work that accelerated only the CNN backbone, the YOLO accelerator handles the complete inference pipeline—pre-processing, all CNN layers (convolution, maxpool, Leaky ReLU, upsample, route), and post-processing—without returning control to the Central Processing Unit (CPU) between steps [30]. Versat-AI adopts the same goal: the ONNX graph is mapped end-to-end onto the accelerator, with the RISC-V CPU managing only scheduling and I/O.

The hardware abstraction is the VRead/VWrite pair. **VRead** is a dual-port embedded memory: one port is driven by an AXI-DMA engine that fetches data from external memory in configurable AXI4 bursts over the 256-bit bus, while the other port is driven by an independent AGU that sequences the buffered words into the arithmetic datapath. **VWrite** is the symmetric counterpart: an internal AGU writes computation results into one port while an external AGU simultaneously drains the other port back to external memory via DMA. Both units operate as ping-pong buffers, sustaining a continuous data stream [30]. Versat-AI inherits both units unchanged and shares them across every supported ONNX operator.

The YOLO accelerator's area footprint (103–147 k LUTs, 832–1248 DSPs on the KU040) is itself constant with respect to the network: its fixed, bandwidth-matched functional-unit count is reused across the entire inference pipeline, the same constant-footprint outcome DeepVersat's stage-chaining and Versat-AI's compiled merge each reach by different means. What it lacks is generality: reaching that reused, bandwidth-matched datapath required a designer to hand-derive it for Tiny-YOLO specifically, so supporting a different network or operator set means repeating that manual design effort from scratch. Versat-AI resolves this by compiling the same reuse strategy automatically for an arbitrary ONNX model: its merge algorithm aggregates the heterogeneous functional units the target operator set requires and reuses both the units and their strictly-necessary routing across operators, producing an accelerator of under 9 k Look-Up Tables (LUTs) regardless of model size without any per-network manual redesign, at the cost of peak throughput that future parallel operator instances will recover.

### *3.2. Open-Source SoC Platform*

Assembling a production-quality SoC from open-source components is a recurring challenge for researchers and educators: the ecosystem offers processors, bus standards, simulation tools, and FPGA support packages, but composing them into a reliable, maintainable platform is rarely as straightforward as the individual tools suggest. Integration failures are common yet almost never published, leaving each new group to rediscover the same obstacles independently. This section documents that experience and the purpose-built platform that emerged from it.

#### 3.2.1. IOb-SoC

IOb-SoC is the open-source RISC-V SoC template that emerged from two prior integration attempts. Blackbird [32] assembled an SoC around the OpenRISC 1000 processor [33], the ORPSoC generator [34], and a Wishbone bus [35], establishing a unified verification strategy spanning RTL simulation with Icarus Verilog [36], cycle-accurate simulation via Verilator [37], and FPGA emulation from the same test program—but stalled when the OpenRISC ecosystem proved too immature and hard dependencies on specific FPGA boards exhausted the project budget. Warpbird [38] replaced OpenRISC with a 64-bit RISC-V Rocket core [39] generated in Chisel [40]/Scala, migrated the bus to AXI4, and made the Rocket Chip infrastructure fully autonomous by adding a Boot ROM and DRAM controller—but the generator proved too monolithic to decompose and adapt, and its debug subsystem could not be integrated into the team's verification flow [41]. The conclusion was that a purpose-built, modular SoC template was the only viable path. It is built around a softcore CPU (such as VexRiscv [42] or Ibex [43]) and uses an AXI4 master bus for external memory access, connecting

 doi:10.20944/preprints202607.1442.v1



to either on-chip block RAM or a third-party DDR controller. A prebuilt peripheral library provides bare-metal drivers for standard components such as UART and timer.

IOb-SoC is now described in Python and generated by the Py2HWSW framework [5], which automates the production of the Verilog sources, bus wrappers, and software drivers for every peripheral or accelerator added to the system.

The framework supports RTL simulation with Icarus [36], Verilator [37], Xcelium (Cadence, including code coverage), Questa (Siemens), and VCS (Synopsys); FPGA synthesis with Vivado (AMD) and Quartus (Altera); ASIC synthesis with Genus (Cadence); and static linting and CDC/RDC checks with ALINT-PRO (Aldec) and SpyGlass (Synopsys), making the same SoC description target-agnostic across the full spectrum from open-source simulation to commercial ASIC sign-off.

#### 3.2.2. Py2HWSW

Py2HWSW [5] is an open-source Python-based framework that acts as an integrator and generator of modular system-on-chip platforms built around RISC-V processors with instantiated peripherals.

System specifications are expressed in Python using high-level abstractions describing processor configuration, memory mapping, and peripheral connectivity. From this description, Py2HWSW automatically generates synthesizable Verilog for system components, memory-mapped interconnect logic, peripheral wrappers, and software support files.

A key capability of Py2HWSW is the integration of heterogeneous components into a coherent system. It connects RISC-V cores with standard peripherals (e.g., UART, timer) and custom hardware accelerators within a unified memory-mapped architecture, enabling seamless interaction between processor software and accelerators such as those compiled by Versat.

## 4. Versat-AI

Versat-AI [44] is a hardware-accelerated inference runtime generator for ONNX models that functions as a Model-to-Runtime compiler. It automatically generates both the specialized hardware accelerator and the embedded software required to execute deep learning inference on a System-on-Chip (SoC). Unlike traditional CGRA-based approaches that focus solely on operator mapping, Versat-AI deploys a complete hardware/software system and its verification environment as shown in Figure 3.

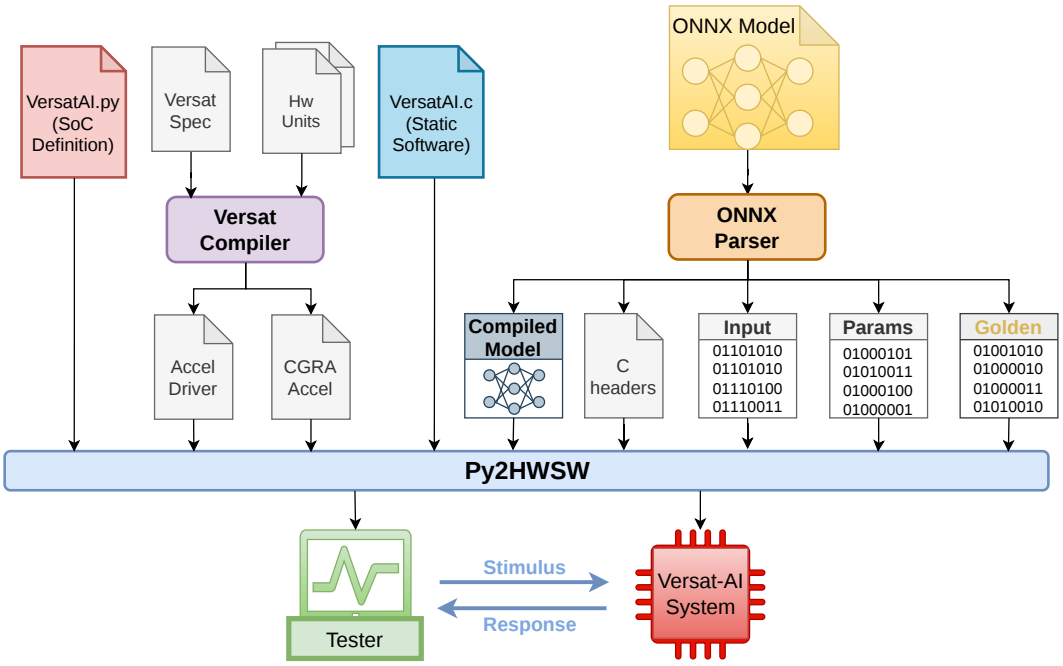


**Figure 3.** Versat-AI workflow.

As shown in Figure 3, three tools cooperate to produce the final system. The Versat Compiler takes the Versat Spec file together with the hardware description of each functional unit and synthesizes

 doi:10.20944/preprints202607.1442.v1



the merged CGRA accelerator. The ONNX Parser takes only the target ONNX model and produces the compiled inference program and its supporting data artifacts. Py2HWSW then assembles the complete system, combining the artifacts generated by the Versat Compiler and the ONNX Parser with the `VersatAI.py` system description and the `VersatAI.c` static inference firmware. During build, Py2HWSW orchestrates system generation, integrating the Versat accelerator with the VexRiscv [42] CPU, memory subsystem, and peripherals to produce the final hardware/software system and test environment.

### *4.1. Versat Compiler*

The Versat Compiler, whose workflow is illustrated in Figure 4, takes two inputs: the Verilog hardware description of each functional unit (e.g. integer adder, floating-point multiplier, etc) and a Versat Spec file, describing a hardware accelerator for each neural network operator in terms of those units. The Unit Registry parses the Verilog units, reading off each one's port list and the latency attribute annotated on its interface, and registers it as a reusable elemental building block. The Specification Parser reads the Versat Spec file and, for each operator module, builds a graph representation of its datapath and configuration logic over the units supplied by the Unit Registry. The Versat Specification and Hardware units database will be detailed in 4.1.1. The per-operator graphs are then flattened to their elemental units, so they can be shared among operators, and merged into a single graph representation describing the Versat Accelerator datapath and configurability, from which the Output Stage generates the CGRA Accelerator description in Verilog and its respective driver in C. The merge strategy will be described in depth in 4.1.3

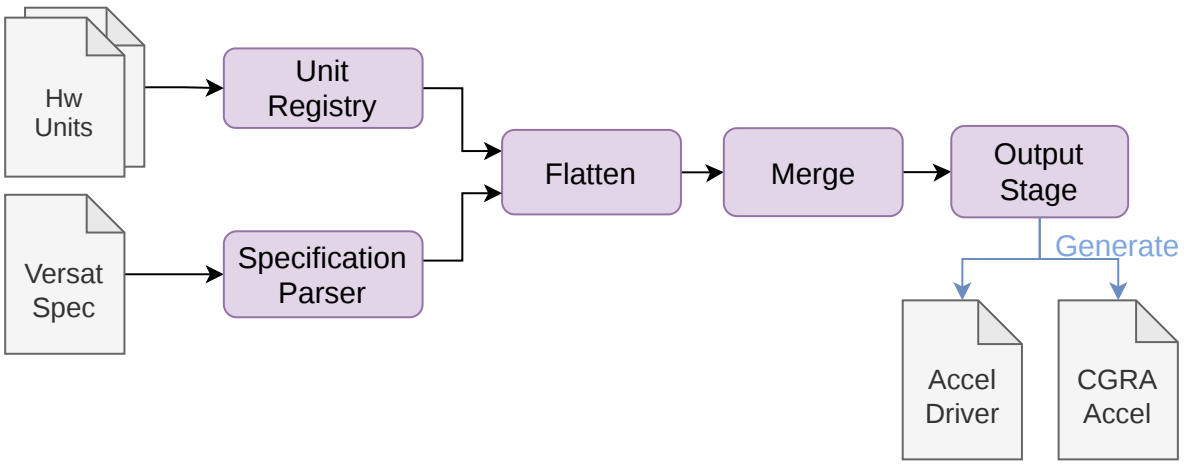


**Figure 4.** Versat compiler workflow.

#### 4.1.1. Versat Specification and Hardware Units

The versat specification contains the description of the operators to be synthesized, working as a higher-level representation of the functionalities to be implemented by the CGRA. Each operator module has a datapath description, in an HDL-like syntax and multiple configuration descriptions in a C-like syntax.

The functional units instantiated by these operator datapaths, and their respective pipeline depths, are described in Table 1. Each unit exposes its own pipeline latency to the compiler via a `versat_latency` Verilog attribute on its output; the compiler uses these per-unit values to statically schedule the merged CGRA datapath and to derive the end-to-end operator depths.

 doi:10.20944/preprints202607.1442.v1



**Table 1.** Functional Unit Description and Latency.

| Unit | Description | Depth (cycles) |
|---|---|---|
| VRead | Streams operands from local memory into the datapath under a configurable access pattern (linear, strided, broadcast, 2-D windowed) | 3 |
| F_Mul | Single-precision floating-point multiplier | 4 |
| F_Add | Single-precision floating-point adder | 5 |
| F_Accum | Single-precision floating-point accumulator; sums a stream of partial results over a configurable window | 4 |
| F_AccumMax | Streaming maximum-reduction unit; compares each incoming sample against a running maximum over a configurable stride | 1 |
| Relu | Rectified-linear-unit activation | 1 |
| Const | Drives a fixed, per-configuration constant value onto the datapath (e.g., an accumulator seed or scale/bias operand); no data-dependent computation, so it adds no pipeline latency | 0 |
| VWrite | Streams datapath results back to local memory under a configurable access pattern; terminal stage of the pipeline | NA |

The operators supported by the Versat-AI CGRA are described in Table 2. The table, lists, for each operator, its name, the principal hardware units its datapath instantiates, and its latency in clock cycles. Because these datapaths are graphs rather than simple chains — units can fan out to, or be shared between, multiple downstream units — the list of units is given as a set rather than as a connection sequence; Section 4.1.3 details how the units are actually wired and subsequently shared across operators.

**Table 2.** Supported ONNX Operators and Hardware Mapping.

| Operator | Units | Depth (cycles) |
|---|---|---|
| Add | VRead, F_Add, VWrite | 8 |
| MatMul | VRead, F_Mul, F_Accum, VWrite | 11 |
| Conv | VRead, F_Mul, F_Add, F_Accum, Const, VWrite | 17 |
| AveragePool | VRead, F_Accum, F_Mul, VWrite | 11 |
| MaxPool | VRead, F_AccumMax, VWrite | 4 |
| BatchNorm | VRead, F_Mul, F_Add, VWrite | 12 |
| ReLU | VRead, Relu, VWrite | 4 |

#### 4.1.2. The MatMul Operator

This section uses the `MatMul` operator as a running example to illustrate the Versat Spec syntax; its specification is given in Listing 1. Given a matrix multiplication $C = A \times B$, where $A$ is $N \times K$ and $B$ (pre-transposed) is $M \times K$, one call to the module computes a single row of the $N \times M$ result matrix: it takes the dimensions $K$ and $M$, a pointer to the corresponding row of $A$, and a pointer to the transposed matrix $B$, and must be invoked $N$ times, once per row of $C$, to produce the complete result.

 doi:10.20944/preprints202607.1442.v1



**Listing 1.** MatMul operator description in the Versat Spec.

```
// Computes one row of C[N*M] = A[N*K] * B[K+M]
module MatMul(){
    VRead ARow;
    VRead BColT;
    F_Mul mat_mpy;
    F_Accum Accum;
    VWrite C;

    ARow -> mat_mpy:0;
    BColT -> mat_mpy:1;
    mat_mpy -> Accum;
    Accum -> C;

    config All(ARow_buf:Buffer, BColT_buf:Buffer, C_buf:Buffer, M:Dyn, K){
        for y 0..M{
            for x 0..K {
                ARow = ARow_buf[x];
                BColT = BColT_buf[y * K + x];
            }
        }
        Accum.strideMinusOne = K - 1;
        for x 0..M {
            C_buf[x / K] = C;
        }
    }
}
```

A module is declared with the `module` keyword, followed by its unit declarations (`Type name;`, e.g. `VRead ARow;`) and a connection graph describing how those units' ports are wired together. Each connection is written `source -> destination`; when no port is given, port 0 is assumed on both ends, and `unit:X` selects port `X` explicitly (so `Accum -> C` is shorthand for `Accum:0 -> C:0`).

A graphical representation of the `MatMul` module is given in Figure 5. This figure also contains the accumulated latency after each unit. The VRead units ARow and BcolT introduce a 3-cycle delay once the accelerator is started. The latency monotonically increases as we go down the Directed Acyclic Graph (DAG) and reaches a maximum at the terminal C VWrite node, from where it is written to memory.

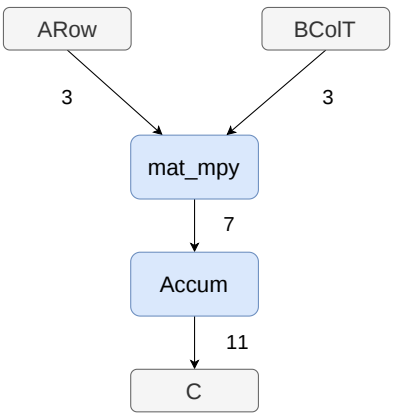


**Figure 5.** Graph corresponding to the MatMul module specification.

After the connections, one or more `config` functions describe how the module is parameterized at runtime. Versat spec files support three function kinds — `config`, `mem`, and `state` — for writing unit configuration, accessing memory-mapped units, and reading unit state back, respectively; only `config` is needed here. Each `config` function is named (the name becomes the generated C function that the

 doi:10.20944/preprints202607.1442.v1



firmware calls, e.g. `All` in the listing) and takes parameters of the form `name:modifier`. In the example, `:Buffer` marks a pointer argument, while integer arguments are either fixed (e.g. *K*) or dynamic `:Dyn` (e.g. *M*). Fixed integer parameters are not checked for hardware size compatibility, whereas dynamic parameters inform the compiler that the hardware size may be exceeded, and the compiler proceeds to automatically fold the loop the number of times necessary to complete the loop with the existing hardware. The distinction between fixed and dynamic parameters is not strictly necessary and the compiler can be upgraded to dispense with it.

The assignment `BColT = BColT_buf[y * K + x]` inside the nested `for y` loop is not unrolled into literal per-cycle values: it symbolically describes the address pattern that `BColT`'s underlying `VRead` unit should generate over time — at loop iteration `(x,y)`, the unit is to produce the value found at `BColT_buf[y * K + x]`, a linear address expression over the loop indices. `ARow_buf` and `BColT_buf` are addresses in external Double Data Rate (DDR) memory, while the VRead and VWrite units each hold a small internal on-chip buffer that is filled from (or drained to) DDR sequentially; the address expression given in the `config` function addresses that fast internal buffer to produce (or consume) one value per cycle on the datapath side. This split exists because DDR only supports efficient sequential/burst access, whereas the arbitrary access patterns Versat operators need (linear, strided, broadcast, 2-D windowed), per Table 1) are only practical against the fast internal buffer. `Accum.strideMinusOne = K - 1;` sets a config wire directly: `F_Accum` counts accumulated values from 0, so `strideMinusOne` is the zero-based terminal count at which it flushes its sum, precomputed here to avoid a hardware subtractor. Finally, `C_buf[x / K] = C` demonstrates a non-linear address expression on the write side: integer division collapses each group of `K` consecutive cycles onto the same internal-buffer address, so only the last (fully accumulated) value of each group is the one that persists once that address is eventually drained to `C_buf` in DDR.

#### 4.1.3. Merge Algorithm

Each operator is specified independently in the Versat Spec, following the structure illustrated for `MatMul` in Section 4.1.2. As noted in Section 4.1.1, functional units can be shared across such operators; this section describes the merge algorithm that performs that sharing.

A `merge` statement in the spec file instructs the compiler to fuse the specified accelerators into a single CGRA. The compiler resolves the merge by identifying structurally compatible hardware units across modules and overlapping them into CGRA configurations, trading off interconnect silicon area for a more complex compiler that can reuse the same hardware units across different operators. This approach realises the strategy of [4] in the novel context of neural network operator mapping, and differs from the original Versat CGRA [27], which used a full crossbar to connect all units.

This unit-sharing scheme builds on prior compatibility-graph merging techniques [45,46]. To build the CGRA, each input operator is first flattened into a graph of its functional units, and the merged graph is grown incrementally: it is initialized as a copy of the first operator, and each remaining accelerator is folded in one at a time. For every pair of units between the incoming operator and the merged graph so far, structural compatibility is evaluated and recorded as a node of a different graph called the compatibility graph; two candidate mappings are connected by an edge only if they can be adopted together, since merging one pair of units can preclude other pairings.

Figure 6 illustrates this process for two example operators, X and Y. X chains `VRead(A)`, `Relu`, `Float_Add` along one input path and `VRead(B)` directly into `Float_Add` along the other; Y instead has a single `VRead(A)` whose output diverges into two paths, one through `Relu` and one without, that reconverge at `Float_Mul`. Both write their result through a `VWrite`. Comparing the two unit-by-unit gives the compatibility graph's nodes: each operator has exactly one `Relu` and one `VWrite`, so each admits only a single candidate mapping, whereas X's two `VRead` instances can each map to Y's single `VRead(A)`, giving two candidate unit mappings. Of the connections in the two datapaths, only the `VRead(A) -> Relu` edge exists in both operators, so it is the sole candidate edge mapping.

 doi:10.20944/preprints202607.1442.v1



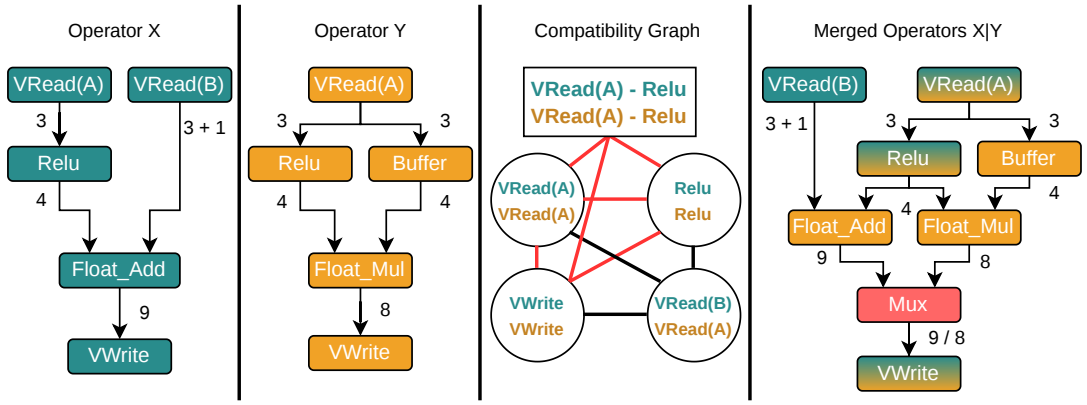


**Figure 6.** Example of two operators, their compatibility graph, and the merged result.

In the compatibility graph, circles denote candidate unit mappings and the rectangle denotes the candidate edge mapping just described; an edge between two candidates indicates they can be adopted together without structural conflict, and the maximum clique — highlighted in red — gives the largest consistent set of shared mappings, with nodes color coded by the input graph they originate from. For X and Y, this clique shares three units: their respective `VRead(A)` instances collapse into one, as do their `Relu` and `VWrite` instances, leaving X's `VRead(B)` as its own unshared instance. This is also why `VRead(A)` is shared over `VRead(B)`: sharing either is structurally possible, but sharing `VRead(A)` avoids inserting a multiplexer, since the `VRead(A) -> Relu` connection already existed as a candidate edge mapping. As with any port left with inputs from more than one configuration, a multiplexer is inserted to select between them: here, to route the output of `Float_Add` or `Float_Mul` into the shared VWrite unit depending on whether X or Y is the currently configured operator. Each configuration's data path is wired to the multiplexer port matching the index of its input operator, an association recorded in the mapping described above so it can be exported as firmware configuration data; the firmware then drives the multiplexer's select input at runtime.

In general, finding the largest set of mutually compatible mappings is a maximum clique [47,48] problem on the compatibility graph, computed as described in Listing 2, so that as many units as possible are safely collapsed into shared instances; units without a compatible match are instantiated anew. This search uses a branch-and-bound algorithm [49], giving it a worst-case complexity of $O(3^{M/3})$ [50] for a compatibility graph with $M$ vertices, bounded by a 10-second timeout to avoid hanging on large or complex consolidation graphs. Merging more than two operators is handled by repeating this fold-in step iteratively, one accelerator at a time.

Merging units that were previously independent can introduce timing mismatches between paths that now feed a shared downstream unit: in the merged graph, edges are annotated with each path's global latency in cycles, and X's path through `VRead(A)` and `Relu` to `Float_Add` is longer than its direct path from `VRead(B)`. When the mismatched paths do not reconverge anywhere else before their common downstream unit — as with X's two independent inputs to `Float_Add` — Versat resolves the difference for free: because `VRead` and `VWrite` units have programmable address generators, the faster path's AGU is simply configured to start issuing (or accepting) addresses later, with no additional buffer hardware needed.

 doi:10.20944/preprints202607.1442.v1



**Listing 2.** Branch-and-bound time-limited MaxClique Algorithm.

```
#define TIME_LIMIT 10.0    // seconds

Clique MaxClique(Graph G){
    Clique best = EmptyClique();
    double startTime = GetTime();
    for (Vertex v : G.vertices){
        VertexSet candidates = Neighbors(v);
        Search(v, candidates, 1, &best, startTime);
    }
    return best;
}

void Search(Vertex v,
            VertexSet candidates,
            int size,
            Clique* best,
            double startTime){
    /* Timeout */
    if (GetTime() - startTime >= TIME_LIMIT)
        return;

    if (Empty(candidates)){
        if (size > Size(*best))
            UpdateBestClique(best);
        return;
    }

    while (!Empty(candidates)){
        Vertex u = FirstCandidate(candidates);
        /* Branch-and-bound */
        if (size + Count(candidates) <= Size(*best))
            return;

        if (size + UpperBound(u) <= Size(*best))
            return;

        VertexSet next = candidates;
        Remove(next, u);
        next &= Neighbors(u);
        Search(u, next, size + 1, best, startTime);
        Remove(candidates, u);
    }
}
```

This AGU-offset trick does not work, however, when the mismatched paths reconverge from a single shared source: in Y, both branches leaving `VRead(A)` are driven by the same AGU, so they cannot be delayed independently before recombining at `Float_Mul`. In such reconvergent cases, a dedicated delay buffer is inserted into the faster (non-`Relu`) branch to re-synchronise the two operands; this buffer is visible in the merged graph in Figure 6, kept there because X never required one.

Because a buffer added for one configuration can itself change the delay seen by others, the mappings and delay calculations are recomputed after every buffer insertion, repeating this correction until the timing of every configuration is consistent; this corrective flow is illustrated in Figure 7. This iteration is needed because delay buffers are runtime-programmable but still impose a hard floor of one cycle each, which is what triggers the loop: inserting a buffer to fix one configuration's timing can only add whole cycles, potentially unbalancing another configuration and forcing a further buffer elsewhere. Once this buffer-insertion process converges and the merged graph's topology is fixed, any additional delay beyond that first cycle is free — it is simply written into the corresponding operator's

 doi:10.20944/preprints202607.1442.v1



configuration, the same AGU-offset mechanism described above, with no further hardware changes required.

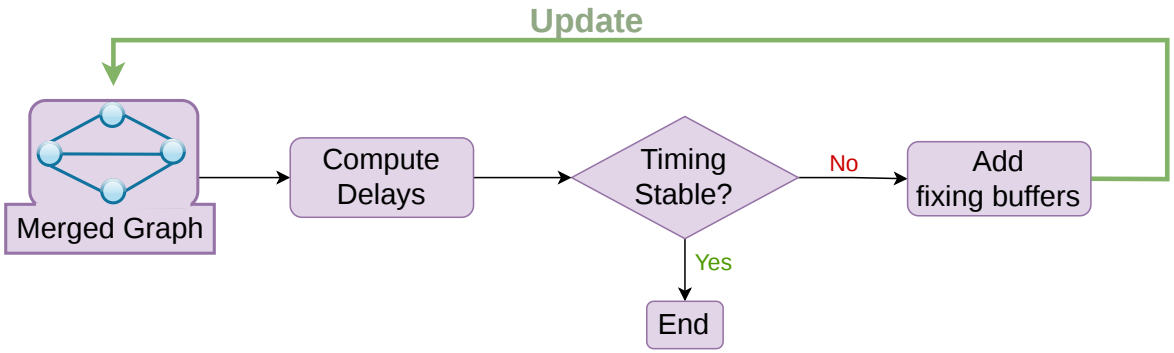


**Figure 7.** Delay correction in the Merge Algorithm.

Throughout the merge process, the compiler records, for each input operator, which merged-graph unit each of its original units was folded into, and the reverse: for each merged-graph unit, which original unit it stands in for under each operator. This mapping is what lets the compiler later recover, for any given operator, how the shared hardware must be configured to reproduce that operator's original dataflow.

With this merge strategy, the accelerator area is determined entirely by the number of unique hardware units required to support the full operator set, rather than the model size. Multiple operators of the same kind are planned, but their number will be tied to the parallelism opportunities, and not to the model size. Therefore, an example 50-layer ResNet uses the same silicon footprint as a 5-layer one.

#### 4.1.4. Dynamic Partial Reconfiguration

Because the merge algorithm collapses multiple operators onto shared hardware, switching between them at runtime requires reconfiguring the shared units' behaviour without resynthesis. This is done through the Versat-AI CSR interface shown in Figure 8: the Config Reg/Config Shadow Reg datapath on the left, and the Read/Compute/Write pipeline schedule on the right.

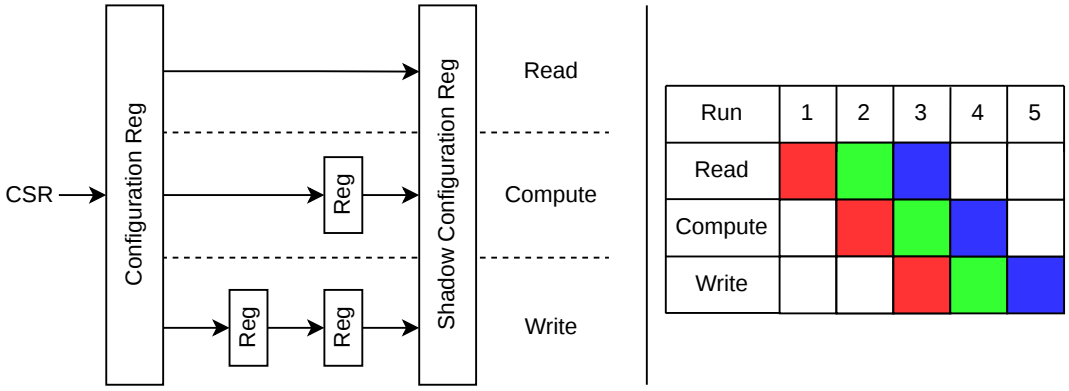


**Figure 8.** Versat-AI's CSR reconfiguration mechanism.

At runtime, the firmware switches operators by writing to the configuration register accessed through this interface. Each functional unit's configuration is split into individually addressable fields, grouped by the pipeline phase they affect — Read, Compute, or Write — as detailed per unit in Table 3. A CSR write lands in the Config Reg and is then propagated, through a per-phase delay chain (no delay for Read, one register for Compute, two for Write — 312, 1194, and 102 bits respectively, 1608 bits in total), into the Config Shadow Reg that actually drives the datapath. Because fields are individually addressable, switching to a new operator only requires rewriting the (typically small) subset of fields

 doi:10.20944/preprints202607.1442.v1



that actually differ from the current configuration, rather than the whole register file; Table 4 lists, for every supported operator, the hardware units it uses, the number of such fields that must be written, and their total bit width.

**Table 3.** Configuration cost per functional unit instance.

| Unit | Fields | Config. Bit widths | | | |
|---|---|---|---|---|---|
| | | Read | Compute | Write | Total |
| VRead | 23 | 104 | 265 | 0 | 369 |
| VWrite | 23 | 0 | 112 | 247 | 359 |
| F_AccumMax | 2 | 0 | 26 | 0 | 26 |
| F_Accum | 2 | 0 | 26 | 0 | 26 |
| Const | 1 | 0 | 32 | 0 | 32 |
| Buffer | 1 | 0 | 10 | 0 | 10 |
| Mux | 1 | 0 | 3 | 0 | 3 |

**Table 4.** Per-operator configuration cost after merging.

| Operator | HW Units | | | | | | | Configuration | |
|---|---|---|---|---|---|---|---|---|---|
| | VRead | VWrite | F_AccumMax | F_Accum | Const | Buffer | Mux | Fields | Bits |
| Add | 2 | 1 | 0 | 0 | 0 | - | - | 69 | 1097 |
| MatMul | 2 | 1 | 0 | 1 | 0 | - | - | 71 | 1123 |
| Conv | 3 | 1 | 0 | 1 | 1 | - | - | 95 | 1524 |
| AveragePool | 1 | 1 | 0 | 1 | 0 | - | - | 48 | 754 |
| MaxPool | 1 | 1 | 1 | 0 | 0 | - | - | 48 | 754 |
| BatchNorm | 1 | 1 | 0 | 0 | 0 | - | - | 46 | 728 |
| Relu | 1 | 1 | 0 | 0 | 0 | - | - | 46 | 728 |
| Post-Merge Mux and Buffers | - | - | - | - | - | 4 | 6 | 10 | 58 |
| **Merged Accelerator** | 3 | 1 | 1 | 1 | 1 | 4 | 6 | 107 | 1608 |

This makes reconfiguration both dynamic and highly partial: it happens at runtime, on demand, per operator switch, touching only the handful of fields that changed — typically a few CSR writes, taking only a few cycles. Crucially, this can happen while the previous configuration is still running: the Config Reg accepts new field values immediately, while the delay chains ensure they only reach the Config Shadow Reg, and thus only take effect, once the corresponding phase is between runs. The right-hand side of Figure 8 illustrates why this is safe with a CPU-pipeline-style schedule: since Read, Compute, and Write are separate, delay-matched stages, up to three runs are in flight at once (e.g. Run 3 reads while Run 2 computes and Run 1 writes), so each phase's fields can be swapped in the moment that phase drains, independently of the other two. Reconfiguration latency is therefore effectively hidden behind the execution of the configuration it replaces.

### *4.2. ONNX Parser*

The ONNX Parser, implemented in Python using the `onnx` [3] and `onnxruntime` [51] libraries, loads the model, runs shape inference to resolve all tensor dimensions, and validates the constituent operators of the neural network Data Flow Graph (DFG) against the hardware capabilities. It emits a binary `CompiledModel struct` of serialized operations and parameters, together with C header artifacts. During parsing, each ONNX operator is matched against the operators listed in the Versat Spec file to determine whether it can be executed on the accelerator or must be handled by the RISC-V CPU. The ONNX Parser also outputs the model's parameters (inputs and weights) and a binary golden reference to validate the generated HW/SW system. Figure 9 illustrates the parser's pipeline, along with the generated artifacts by each stage.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 20 July 2026 doi:10.20944/preprints202607.1442.v1



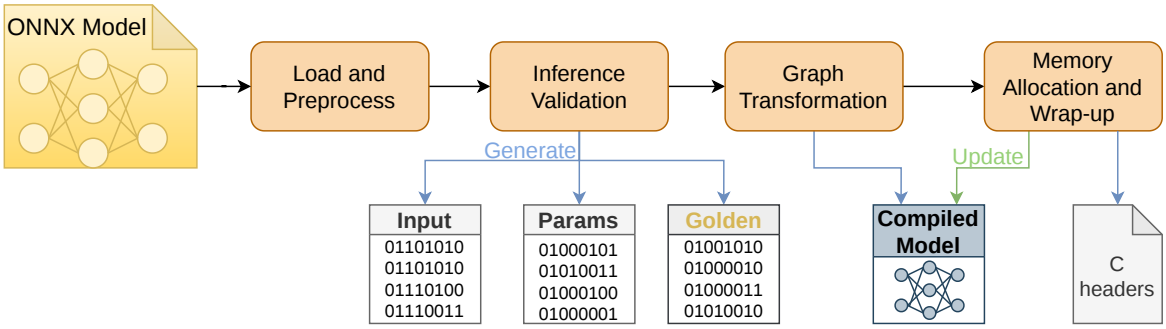


**Figure 9.** Multi-stage ONNX parser structure.

#### 4.2.1. Load & Preprocess

The pipeline begins by loading the ONNX `protobuf` files and running the standard ONNX shape-inference pass to resolve all tensor dimensions. Initializers are separated from live graph inputs (as required by ONNX IR 4 and above), and the model structure is validated against the ONNX specification. No compilation artifacts are produced at this stage.

#### 4.2.2. Inference Validation

Before a model is trusted as a reference for hardware validation, the parser first checks that the model itself is sound, if the model is distributed with its own bundled test vectors: a sample input together with its expected output, as computed independently of Versat-AI. The preprocessed model is executed through ONNX Runtime on that bundled input to produce a floating-point golden reference, which is compared against the bundled expected output with a relative tolerance of $10^{-5}$; any mismatch aborts the pipeline, since a model ONNX Runtime cannot reproduce correctly on its own test data cannot serve as a trustworthy reference for validating Versat-AI's hardware output later on. Only once this check passes is the golden reference computed here carried forward, alongside the model, for that later hardware validation.

#### 4.2.3. Graph Transformation

Every ONNX operator supported by Versat-AI must provide a software-only implementation, executable on the RISC-V CPU, and may additionally provide a CGRA implementation via the Versat Spec file. Each node in the ONNX DFG is matched against this operator registry: if a CGRA implementation is available, the node is scheduled onto the accelerator; otherwise, if only a software implementation exists, the node falls back to CPU execution; operators with neither cause an immediate compilation error, giving the user a clear indication of which operators are entirely unsupported. For nodes that are scheduled, tensor shapes, operator attributes (strides, padding, dilation, etc.), and data-source classifications (model weight, live activation, or another node's output) are extracted and assembled into an intermediate representation that drives the subsequent stages.

At this stage we also perform graph optimizations that change data format into an implementation that is easier to accelerate. The end result is the same thus ensuring that these optimizations do not affect the correctness of results.

One example is related to the MatMul operation. In a normal implementation of MatMul, iterating the columns of the right side matrix is costly since they are stored in row-major order. MatMul is an operation that regularly uses constants as the right side matrix. This gives us an opportunity to implement a simple optimization: instead of iterating the columns, we can iterate the rows of the transposed matrix.

By storing the constant already transposed and by setting an attribute on the operator that instructs that the right side matrix is already transposed, we can use a faster MatMul implementation while producing the same results.

 doi:10.20944/preprints202607.1442.v1



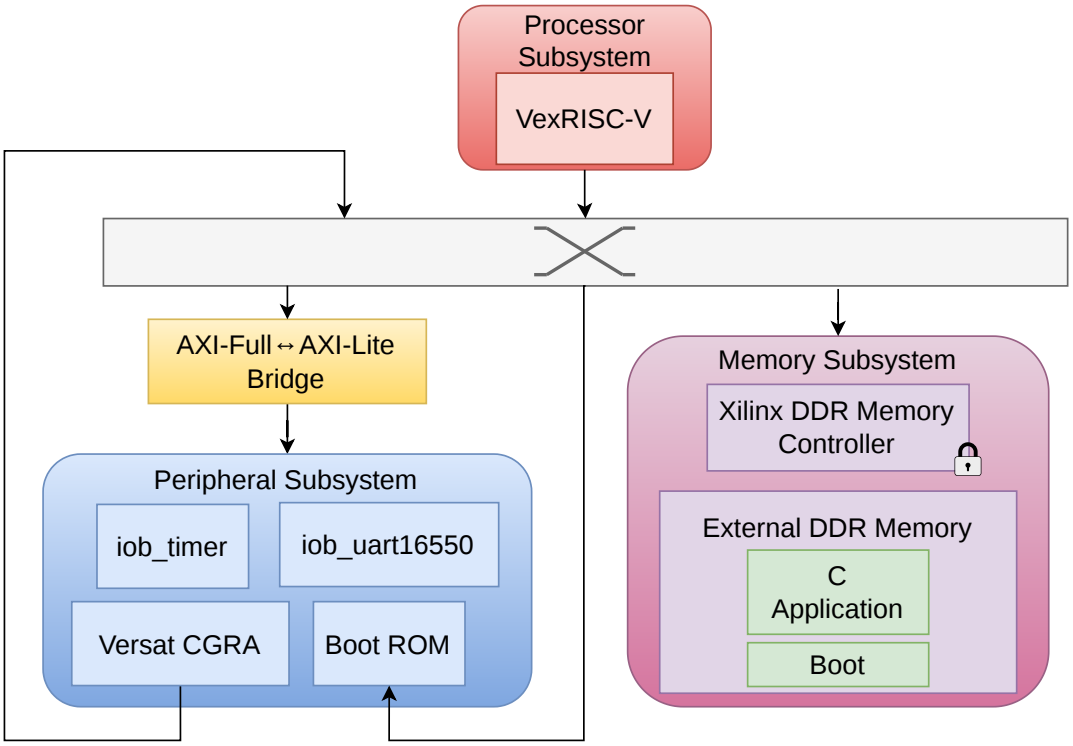


**Figure 10.** Versat-AI System

#### 4.2.4. Memory Allocation and Wrap-Up

A liveness pass determines, for each intermediate tensor, the last graph node that consumes it. A greedy interval-fitting algorithm then packs live tensors into a single scratch-memory region, reusing space as soon as a tensor's lifetime ends. All allocations are 4-byte aligned to match the float32 element size. Permanent outputs are placed in a separate, sequentially allocated output region. The resulting offsets are recorded in the compiled model and used by the firmware runtime to address the on-chip memory.

The compiled model is serialized into a binary `CompiledModel` struct consumed by the firmware runtime, together with a model-independent C header that defines operator-type enumerations, per-operator parameter structs with variable-length trailing arrays, and accessor macros. The weight tensors and the floating-point golden reference generated in the validation stage are also written to image, providing the firmware loader with all data required to initialize the accelerator. Once inference completes, the firmware compares Versat-AI's computed output element-wise against this embedded golden reference, using the same $10^{-5}$ relative tolerance as the earlier ONNX Runtime check, and reports a pass/fail result. This validation runs identically whether the firmware executes in host-CPU emulation or on the physical FPGA target (Section 4.3.2), since both run the same code against the same golden reference.

### *4.3. SoC Integration Framework*

The `VersatAI.py` module describes the complete SoC as a Py2HWSW `iob_core` subclass, combining module definitions—ports, internal wires, and instantiated child modules—with instance properties such as parameter overrides and port-to-wire mappings. This allows the entire system to be composed from reusable building blocks without hand-written integration glue. The generated Versat-AI system is illustrated in Figure 10.

 doi:10.20944/preprints202607.1442.v1



4.3.1. CSR and Software Artifact Co-Generation.

For every peripheral instantiated in the system description, Py2HWSW generates a matched hardware/software pair from the same Python source. On the hardware side it produces a Verilog CSR module exposing each control register over the peripherals' bus. On the software side it produces a C header with macro definitions and a bare-metal driver with read/write accessor functions. For Versat-AI, the CSR block exposes the CGRA configuration register – operator-select, AGU parameters, and run-control signals – whose addresses and field layouts are kept in exact alignment with the firmware that the ONNX parser generates, since both are derived from the same Python description. The firmware runtime assembles these CSR writes at inference time from the `CompiledModel` struct emitted by the parser, configuring the CGRA for each operator in the execution schedule without any manually written glue code.

4.3.2. Unified Build and Emulation

Py2HWSW's hierarchical Makefile exposes simulation, FPGA synthesis, and host emulation from a single configuration .py file. An emulation mode compiles the SoC firmware for the host CPU and replaces hardware CSR accesses with memory-mapped reads and writes into a Verilator model, providing a co-simulation environment in which the same application runs on both host emulation and the FPGA target. This target-agnostic flow eliminates separate board support packages and ensures that simulation-passing firmware is byte-for-byte identical to what executes on the Kintex Ultrascale KU040.

## 5. Experimental Results

Versat-AI was evaluated end to end on a Xilinx Kintex Ultrascale FPGA (XCKU040) running at 100 MHz, using four MLPerf Tiny [6] inference tasks as a representative cross-section of edge workloads spanning image classification, anomaly detection, keyword spotting, and person detection. The evaluation is organised to answer five questions in turn. Section 5.1 asks whether the accelerator's silicon footprint is really independent of the evaluated models, as the compiled unit-merge strategy claims. Section 5.2 checks that the compiler's own generated software is a fair, unhandicapped CPU baseline before that baseline is used to measure accelerator speedup. Section 5.3 then reports that speedup and situates it against related CGRA and dataflow accelerators. Section 5.4 presents a power and energy analysis, translating the measured speedup into energy per inference under the system's actual power budget. All of these results describe the current, single-operator-instance phase of the project, in which each operator currently runs on a single physical datapath instance rather than several in parallel; Section 5.5 closes by laying out the planned upgrade to parallel operator instancing and projecting its impact on both speedup and energy under explicitly stated, idealized assumptions.

### *5.1. FPGA Resource Utilization*

The resources utilized by the generated system are presented in Table 5. In this version, a single instance of each floating-point operator is used. However, keep in mind that the plan is to have multiple fixed-point instances to scale the performance while keeping the overall footprint predictable and modular.

Compared to some related works, the Versat-AI accelerator footprint is independent of model size. FINN allocates a dedicated hardware pipeline per network layer, so LUT and BRAM usage grows with the number of layers [20]. Gemmini employs a reusable systolic array that amortizes computation across layers, but it primarily targets ASIC flows and requires a large FPGA to host the full RISC-V system alongside the array [16]. Versat-AI's unit-merge strategy instead collapses all operator variants into a single CGRA, holding the accelerator to 8,763 LUTs and 4 DSPs for the full operator set regardless of model size, a key advantage for resource-constrained edge deployments.

 doi:10.20944/preprints202607.1442.v1



**Table 5.** Resource utilization for the XCKU040 FPGA.

| | Component | LUTs | FFs | DSPs | BRAMs |
|---|---|---|---|---|---|
| Versat-AI | Versat Accelerator | 8,763 | 9,833 | 4 | 202 |
| | VexRISC-V | 3,062 | 2,571 | 4 | 5 |
| | DDR4 Controller | 10,728 | 13,740 | 3 | 26 |
| | Other periph. & bus | 4,984 | 4,361 | 0 | 1 |
| | **Total** | **27,537** | **30,505** | **11** | **234** |
| Base SW | VexRISC-V + FPU | 5,074 | 4,503 | 7 | 5 |
| | DDR4 Controller | 10,714 | 13,740 | 3 | 26 |
| | Other periph. & bus | 2,855 | 4,343 | 0 | 1 |
| | **Total** | **18,643** | **22,586** | **10** | **32** |

### *5.2. Software Generation Quality*

Before using our own compiler's generated software as the baseline for accelerator speedup, we first check that it is a competent, representative CPU implementation rather than an artificially weak one—since an accelerator's apparent speedup is only meaningful if measured against a fair software baseline. Table 6 compares the execution time of C code generated by our ONNX parser (Base SW) against ONNX2C [52], an independent, third-party transpiler that converts ONNX models directly to C, on the same VexRISC-V CPU for the four evaluated benchmarks; the Ratio column is the ONNX2C time divided by the Base SW time, so values above 1 indicate Base SW is faster.

**Table 6.** Software generation quality: execution time (seconds), ONNX2C vs. Base SW.

| Benchmark | ONNX2C | Base SW | Ratio |
|---|---|---|---|
| ResNet / CIFAR-10 | 5.08 | 6.59 | 0.77 |
| Dense / ToyCar | 0.18 | 0.09 | 2.00 |
| DS-CNN / Speech | 1.45 | 2.05 | 0.71 |
| MobileNet / VWW | 10.33 | 5.85 | 1.77 |

Base SW is faster than the independent ONNX2C baseline on two of the four benchmarks and slower on the other two, with no consistent bias in either direction. This indicates Base SW is a reasonably competent, unhandicapped software implementation rather than a deliberately weak strawman, which is the property that matters for using it as the baseline in the speedup comparison that follows: unlike a comparison against ONNX2C, where an accelerator's apparent speedup could partly reflect ONNX2C's own code generation choices rather than the accelerator's actual contribution, comparing against Base SW isolates the effect of the accelerator alone, since both the software-only and accelerated paths share the same compiler-generated software backbone.

### *5.3. Performance*

A subset of the `MLPerf Inference:Tiny` benchmark [6] was used for evaluation: image classification with ResNet [53] on CIFAR-10 [54], anomaly detection with a dense autoencoder on the ToyADMOS dataset [55], keyword spotting with DS-CNN [56] on Speech Commands [57], and person detection with MobileNet [58] on Visual Wake Words [59]. Table 7 reports the inference latency for CPU-only execution using our ONNX-parser (Base SW) and for Versat-AI. Both use the same VexRISC-V core, but the CPU-only configuration requires a hardware FPU to perform the inference in software, which roughly doubles the CPU's resource footprint compared to the FPU-less VexRISC-V used along-

 doi:10.20944/preprints202607.1442.v1



side the accelerator in the Versat-AI system, where the accelerator itself performs all floating-point computation. The reported speedup is the Base SW time divided by the Versat-AI time.

**Table 7.** Execution time (seconds) and speedup.

| Benchmark | Base SW | Versat-AI | Speedup |
|---|---|---|---|
| ResNet / CIFAR-10 | 6.59 | 2.91 | 2.3× |
| Dense / ToyCar | 0.09 | 0.01 | 9× |
| DS-CNN / Speech | 2.05 | 0.61 | 3.4× |
| MobileNet / VWW | 5.85 | 1.07 | 5.5× |

Since Versat-AI maps operators onto a fixed set of merged hardware units, accelerator area is independent of network size: the same fixed-footprint accelerator instance, with no resynthesis or area change, was used to run all four evaluated models.

Versat-AI's own speedup can also be set alongside related CGRA and dataflow accelerators, though not as a head-to-head ranking: what is compared here is the present single-operator-instance Versat-AI, which instantiates a single physical datapath per operator, against these works' own reported figures, some of which use parallel operator replication. Within that scope, CONVOLVE's CGRA path reaches an average 3.6× speedup over its RISC-V software baseline [22]. Despite its single-operator-instance accelerator, Versat-AI matches or exceeds that figure on two of four benchmarks (9× on ToyCar, 5.5× on MobileNet), against 2.3× on ResNet and 3.4× on Speech. The lower speedup on ResNet follows from its higher arithmetic intensity: standard $3 \times 3$ convolutions reuse each loaded weight many times, so its CPU baseline is closer to compute-bound and the bandwidth-disciplined accelerator has less memory traffic to exploit. The autoencoder and MobileNet's depthwise-separable convolutions reuse weights far less, placing them in the memory-bandwidth-bound regime this design targets.

SNAFU and RipTide report 9.9× and 6.2× improvements respectively, but on general dataflow programs rather than MLPerf Tiny tasks, and without a direct ONNX front end or full SoC generation [10,11]. FINN and Gemmini operate on quantized integer models and target different benchmark suites, making direct latency comparison impractical: FINN's hardware footprint scales with model depth and width, and while Gemmini's Chipyard integration and CGRA4ML [18] each produce a complete SoC around their accelerator, both build it around a fixed, designer-sized array template rather than one a compiler derives automatically from an arbitrary ONNX model—and CGRA4ML's front end further accepts only QKeras-trained models rather than a standard interchange format [16,18,20].

Closing the remaining throughput gap without giving up this generality is the target of the planned multi-operator-instances parallelism projected in Section 5.5, which would let the compiler reach automatically, for any ONNX model, the peak throughput that YOLO-Versat's hand-crafted, network-specific pipeline achieves by manual design.

### *5.4. Power and Energy Analysis*

Post-implementation power analysis using Xilinx Vivado estimates the complete SoC at 1.96 W at 100 MHz, broken down in Table 8.

**Table 8.** Post-implementation power breakdown (Vivado estimate, 100 MHz).

| Component | Base SW (W) | Versat-AI (W) | Increase |
|---|---|---|---|
| Versat Accelerator | – | 0.568 | – |
| DDR4 Controller | 0.804 | 0.809 | 1.0× |
| VexRISC-V CPU (+FPU for Base SW) | 0.058 | 0.037 | 0.6× |
| Peripherals & bus | 0.033 | 0.046 | 1.4× |
| Total Dynamic | 0.895 | 1.460 | 1.6× |
| **Total Power** | **1.383** | **1.959** | **1.4×** |

 doi:10.20944/preprints202607.1442.v1



The DDR4 controller accounts for 41% of total system power, reflecting the memory-bandwidth-bound nature of floating-point Deep Neural Network (DNN) inference. The Versat accelerator itself draws only 0.65 W—33% of system power—despite performing all operator-level computation.

Because system power is approximately constant within each configuration, energy per inference scales directly with latency, computed as $E = P_{\text{SoC}} \times t$, using the measured total power from Table 8 for each configuration (1.383 W for Base SW, 1.959 W for Versat-AI). Table 9 reports the resulting energy per inference for both configurations.

**Table 9.** Energy per inference.

| Benchmark | Base SW (J) | Versat-AI (J) | Reduction |
|---|---|---|---|
| ResNet / CIFAR-10 | 9.13 | 5.70 | 1.6× |
| Dense / ToyCar | 0.125 | 0.020 | 6.4× |
| DS-CNN / Speech | 2.84 | 1.20 | 2.4× |
| MobileNet / VWW | 8.10 | 2.10 | 3.9× |

The energy reductions are smaller than the raw speedup ratios because the software-only SoC also draws less power than the full Versat-AI SoC (Table 8): the energy reduction equals the speedup scaled by the inverse of the 1.4× power increase shown there (≈ 0.71). Even so, energy reduction still tracks latency reduction directly within each configuration, and because the merge strategy ties the accelerator's footprint to the operator vocabulary rather than model size, these savings hold across all evaluated models regardless of size. An ASIC implementation at a sub-28 nm node would reduce the accelerator contribution substantially: prior work on dataflow accelerators of comparable complexity reports active power under 10 mW at equivalent throughput [10,11], suggesting that the dominant system-level power term would shift from compute to memory I/O even more strongly. Future fixed-point quantisation will reduce the DDR4 traffic directly, targeting an overall system energy reduction beyond the speedup gains alone.

### *5.5. Projected Impact of Parallel Operator Instancing*

The results above describe the current, single-operator-instance phase of the project, in which each operator runs on a single physical datapath instance over a 32-bit memory port, processing one data element per cycle. The planned next step keeps each operator's datapath unchanged but instantiates it $N$ times in parallel, so that $N$ copies of the same operation apply concurrently to different data—the same granularity as YOLO-Versat's own compute array [30]. The limits are memory bandwidth and chip area, and both point to substantial headroom beyond today's single instance.

Versat-AI's target board (Avnet AES-KU040-DB-G) provides a 32-bit DDR4 interface at 1600 Mbps, a 6.4 GB/s theoretical peak; applying an 80% sustained-access derating (justified by the VRead/VWrite units' sequential-only access pattern) gives an estimated 5 GB/s of usable bandwidth. YOLO-Versat's own measured design offers a concrete scaling anchor on the same FPGA family: its 832-MAC array sustains 228 GOPS while its own weight traffic alone (8.8 M parameters, 17.6 MB at 16 bits) requires well under 1 GB/s over its measured 24.4 ms execution time [30], and that same design occupies only 44% of its FPGA's DSP budget. Scaling from that operating point, Versat-AI's available bandwidth and chip area could plausibly support on the order of $10^3$ parallel MAC units; we use $N \approx 1{,}500$ as a round working estimate, comfortably within the ∼1,900-MAC DSP budget the same device family implies.

Table 10 summarises what that scale-up would mean, alongside the assumption behind each number. Throughput assumes all 1,500 MACs operate every cycle at the current 100 MHz clock ($1{,}500 \times 100\,\text{MHz} \times 2 = 300\,\text{GOPS}$), an idealised upper bound with no scheduling or synchronisation overhead. Power assumes each 16-bit fixed-point MAC costs 4.0 mW, YOLO-Versat's own resource-proportional estimate (its reported 3.87 W total SoC power, allocated across components by their share of LUTs, FFs, BRAMs, and DSPs, attributes 3.5–4.6 mW per MAC to its compute array [30]); $1{,}500 \times 4.0\,\text{mW} = 6.0\,\text{W}$ of MAC power is added to Versat-AI's own measured 1.31 W fixed overhead (DDR4 controller, CPU, peripherals, Table 8, assumed unchanged) for an estimated 7.3 W total.

 doi:10.20944/preprints202607.1442.v1



**Table 10.** Projected impact of parallel operator instancing (idealized, 16-bit fixed-point; not measured).

| | Current | Projected |
|---|---|---|
| Parallel MAC units | 1 | ∼1,500 |
| Throughput | 0.2 GOPS | ∼300 GOPS |
| System power | 1.96 W | ∼7.3 W |

These figures describe a hardware-capability ceiling, not a measured or directly inferable end-to-end task speedup: they show how many MACs the board's bandwidth and the device's area could sustain, not how much faster any specific benchmark would run, which depends on what fraction of its runtime is spent in bandwidth- and compute-parallelisable operators. The underlying bandwidth and power ratios are anchored entirely to YOLO-Versat's own measurements and have not yet been demonstrated on Versat-AI's own merged-unit datapath; realising them is the target of future work. Even so, they illustrate that today's single-instance results are an early step on a design with substantial, quantifiable room left to grow.

## 6. Conclusions

This paper presented Versat-AI, an open-source ONNX-to-SoC compiler that automatically derives a complete CGRA-based edge inference system—accelerator, firmware, and application software—from an arbitrary ONNX model. The central technical contribution is the application of the hardware merge strategy [4] to neural network operator mapping: structurally compatible ONNX operators are collapsed into a physical CGRA instance, so the accelerator's hardware directly reflects the operator vocabulary the input model actually uses, rather than a designer-fixed, model-independent template or a hardware pipeline allocated per layer. Integrated with the Py2HWSW [5] SoC generator, the system produces a complete synthesisable RISC-V SoC—RTL, memory-mapped interconnects, firmware drivers, and application software—from a single ONNX input, eliminating the manual effort of hardware/software co-design for edge inference.

The accelerator was validated on a Xilinx Kintex Ultrascale FPGA (XCKU040) at 100 MHz, supporting eight ONNX operators in 8,763 LUTs, 9,833 FFs, 4 DSPs, and 202 BRAMs—a footprint that remained unchanged across all evaluated models regardless of size. The complete SoC totals 27,537 LUTs and 30,505 FFs at 1.96 W, of which the accelerator accounts for only 0.65 W. The merge strategy ties both silicon footprint and power to the operator vocabulary rather than model size, so energy per inference scales with the speedup achieved on the four MLPerf Tiny tasks [6] evaluated: 2.3× to 9× lower than a software-only baseline. These are the results of the current, single-operator-instance phase of the project, in which each operator currently runs on a single physical datapath instance over a 32-bit memory port, and they are evidence that the automatic derivation itself is sound. The compiler's next step is to extend that same automation from sizing the operator vocabulary to sizing per-operator parallel instancing and bandwidth, following the bandwidth-matched scaling principle already demonstrated, by hand, in YOLO-Versat, this accelerator's own design lineage. Section 5.5 works out what that extension would mean in practice, grounded in YOLO-Versat's own measured bandwidth-sharing and power data. Among related flows, FINN [20] scales resource usage with network depth and Gemmini [16] targets ASIC flows on high-density FPGAs; CGRA4ML [18] compiles a complete taped-out SoC around a CGRA, but from a fixed array template and a QKeras-specific front end rather than an unmodified ONNX model; CONVOLVE [22] reports a 3.6× average speedup on its CGRA path, a figure Versat-AI's single-operator-instance accelerator matches or exceeds on two of four benchmarks. None of these, however, derives a complete SoC—accelerator, firmware, and application software—automatically from an unmodified ONNX model with a footprint tied to the model's own operator vocabulary.

The current implementation operates on floating-point data with single operator instances. Future work will add fixed-point quantisation to reduce memory bandwidth and improve energy efficiency [1]—cutting the DDR4 traffic that currently dominates system power—widen the accelera-

 doi:10.20944/preprints202607.1442.v1



tor's 32-bit memory port toward the bandwidth-matched interfaces YOLO-Versat already validated, and instantiate $N$ parallel copies of each operator's own datapath, applying the same operation to different data concurrently, with the compiler automatically sizing their number to each model's own widest layer, scaling throughput while preserving the tie between footprint and operator vocabulary rather than model size. The design will also be validated through an ASIC flow targeting a sub-watt embedded implementation. Added compute parallelism should particularly benefit compute-bound workloads such as ResNet's standard convolutions, where speedup was lowest in the present evaluation. This parallelism is intended to let the compiler reach, automatically and for any supported ONNX model, the throughput that YOLO-Versat's hand-crafted, network-specific pipeline achieves by manual design.

Finally, the background section traced the two-decade evolution of reconfigurable accelerators—from SideWorks and Versat through DeepVersat and the YOLO accelerator—that shaped Versat-AI's design, and documented the open-source SoC platform development, including integration attempts with the OpenRISC ecosystem [33] and the Rocket Chip generator [39], and the design decisions that led to IOb-SoC and Py2HWSW.

**Acknowledgments:** This project was mainly funded by the NLnet NGI0 Core Fund, with support from EC's Next Generation Internet programme, grant agreement No 101092990. The project was partially funded by the A-IQ-Ready project/Chips JU grant agreement No. 101096658. João Barreiros Rodrigues and Jaime Aguiar acknowledge FCT's support through grant BI | 2025/772– Proj. 2023.16747.ICDT (SMARTChip, DOI: https://doi.org/10.54499/2023.16747.ICDT, Ref[a] LISBOA2030-FEDER-00692100), and projects UID/6486/2025, UID/PRR/6486/2025, UID/PRR2/06486/2025, UID/50021/2025, UID/PRR/50021/2025.

## References

1. Sze, V.; Chen, Y.H.; Yang, T.J.; Emer, J.S. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE* **2017**, *105*, 2295–2329. https://doi.org/10.1109/JPROC.2017.2761740.
2. Liu, L.; Zhu, J.; Li, Z.; et al. A Survey of Coarse-Grained Reconfigurable Architecture and Design: Taxonomy, Challenges, and Applications. *ACM Computing Surveys* **2019**, *52*. https://doi.org/10.1145/3357375.
3. ONNX Community. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx, 2019. GitHub repository, accessed June 2026.
4. De Sousa, J.T.; Martins, V.M.G.; Lourenco, N.C.C.; Santos, A.M.D.; Ribeiro, N.G.D.R. Reconfigurable coprocessor architecture template for nested loops and programming tool, 2012. US Patent 8,276,120.
5. IObundle. Py2HWSW. https://github.com/IObundle/py2hwsw, 2026.
6. Banbury, C.; Reddi, V.J.; Torelli, P.; Jeffries, N.; Kiraly, C.; Holleman, J.; Montino, P.; Kanter, D.; Warden, P.; Pau, D.; et al. MLPerf Tiny Benchmark. In Proceedings of the Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track, 2021.
7. Ebeling, C.; Cronquist, D.C.; Franklin, P. RaPiD—Reconfigurable Pipelined Datapath. In Proceedings of the Proceedings of the 6th International Workshop on Field-Programmable Logic and Applications (FPL), 1996, pp. 126–135. https://doi.org/10.1007/3-540-61730-2_35.
8. Mei, B.; Vernalde, S.; Verkest, D.; De Man, H.; Lauwereins, R. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In Proceedings of the Proceedings of the 13th International Conference on Field Programmable Logic and Applications (FPL), 2003, pp. 61–70. https://doi.org/10.1007/978-3-540-45234-8_7.
9. Wijerathne, D.; Li, Z.; Huynh, T.; Mitra, T. Accelerating Edge AI with Morpher: An Integrated Design, Compilation and Simulation Framework for CGRAs. arXiv:2309.06127, 2023.
10. Gobieski, G.; Ghosh, S.; Heule, M.; Mowry, T.C.; Nowatzki, T.; Beckmann, N.; Lucia, B. RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture. In Proceedings of the Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO '22), 2022, pp. 546–564. https://doi.org/10.1109/MICRO56248.2022.00046.
11. Gobieski, G.; Atli, A.O.; Mai, K.; Lucia, B.; Beckmann, N. Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In Proceedings of the Proceedings of the 48th Annual International Symposium on Computer Architecture (ISCA '21), 2021, pp. 1027–1040. https://doi.org/10.1109/ISCA52012.2021.00084.

 doi:10.20944/preprints202607.1442.v1




12. Carpentieri, G.; Ottavi, L.; Rossi, D.; Conti, F. Performance Evaluation of Acceleration of Convolutional Layers on OpenEdgeCGRA, 2024, [2403.01236].
13. Flamand, E.; Rossi, D.; Conti, F.; Loi, I.; Pullini, A.; Rotenberg, F.; Benini, L. GAP-8: A RISC-V SoC for AI at the Edge of the IoT. In Proceedings of the Proceedings of the 29th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), 2018. https://doi.org/10.1109/ASAP.2018.8445101.
14. Garofalo, A.; Rusci, M.; Conti, F.; Rossi, D.; Benini, L. PULP-NN: Accelerating Quantized Neural Networks on Parallel Ultra-Low-Power RISC-V Processors. *Philosophical Transactions of the Royal Society A* **2020**, *378*. https://doi.org/10.1098/rsta.2019.0155.
15. Schiavone, P.D.; Machetti, S.; Butera, S.; Bettinelli, I.; Nadalini, D.; Tortorella, Y.; Rossi, D.; Atienza, D. X-HEEP: An Open-Source, Configurable and Extendible RISC-V Microcontroller for the Exploration of Ultra-Low-Power Edge Accelerators, 2024, [2401.05548].
16. Genc, H.; Kim, S.; Amid, A.; Haj-Ali, A.; Iyer, V.; Padhi, P.; Phothilimthana, P.M.; Nikolic, B.; Katz, R.H.; Asanovic, K. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In Proceedings of the Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC '21). IEEE, 2021, pp. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216.
17. Amid, A.; Biancolin, D.; Gonzalez, A.; Grubb, D.; Karandikar, S.; Liew, H.; Magyar, A.; Mao, H.; Ou, A.; Pemberton, N.; et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* **2020**, *40*, 10–21. https://doi.org/10.1109/MM.2020.2996616.
18. Abarajithan, G.; Ma, Z.; Munasinghe, R.; Restuccia, F.; Kastner, R. CGRA4ML: A Hardware/Software Framework to Implement Neural Networks for Scientific Edge Computing. *arXiv preprint arXiv:2408.15561* **2024**.
19. Chen, T.; Moreau, T.; Jiang, Z.; Zheng, L.; Yan, E.; Shen, H.; Cowan, M.; Wang, L.; Hu, Y.; Ceze, L.; et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In Proceedings of the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), Carlsbad, CA, USA, 2018; pp. 578–594.
20. Umuroglu, Y.; Fraser, N.J.; Gambardella, G.; Blott, M.; Leong, P.; Jahre, M.; Vissers, K. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In Proceedings of the Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM, 2017, FPGA '17, pp. 65–74.
21. Duarte, J.; Han, S.; Harris, P.; Jindariani, S.; Kreinar, E.; Kreis, B.; Ngadiuba, J.; Pierini, M.; Rivera, R.; Tran, N.; et al. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation* **2018**, *13*, P07027. https://doi.org/10.1088/1748-0221/13/07/P07027.
22. Alladi, S.; Lopoukhine, A.; Alexandris, G.; Nardi-Dei, A.; Reddy, R.R.; Lamprakos, C.P.; Chaidos, P.; Maras, A.; Ros, A.; Grosser, T.; et al. Optimize edge AI processing through innovative compilation techniques. In Proceedings of the 2026 Design, Automation & Test in Europe Conference (DATE). IEEE, 2026, pp. 1–7.
23. AMD Inc.. *Vitis AI User Guide (UG1414)*. AMD Inc., 2024. Accessed: 2025.
24. Coreworks SA. SideWorks Reconfigurable DSP Data Engine and FireWorks 32-bit RISC Processor. https://www.design-reuse.com/news/19500/digital-signal-processing-dsp-core.html, 2008. Design-Reuse product announcement, accessed June 2026.
25. Coreworks SA. First Reconfigurable Licensable Core to Be Independently Benchmarked with the BDTI DSP Kernel Benchmarks. https://www.design-reuse.com/news/19919/mid-grain-array-reconfigurable-architecture.html, 2009. Design-Reuse press release, accessed June 2026.
26. Lopes, J.D.; de Sousa, J.T. Versat, a Minimal Coarse-Grain Reconfigurable Array. In Proceedings of the High Performance Computing for Computational Science – VECPAR 2016; Dutra, I.; Camacho, R.; Barbosa, J.; Marques, O., Eds., Cham, 2017; Vol. 10150, *Lecture Notes in Computer Science*, pp. 174–187. https://doi.org/10.1007/978-3-319-61982-8_17.
27. Lopes, J.D.; Véstias, M.P.; Duarte, R.P.; Neto, H.C.; de Sousa, J.T. Coarse-Grained Reconfigurable Computing with the Versat Architecture. *Electronics* **2021**, *10*, 669. https://doi.org/10.3390/electronics10060669.
28. Singh, H.; Lee, M.H.; Lu, G.; Kurdahi, F.; Bagherzadeh, N.; Chaves Filho, E. MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Computation-Intensive Applications. *IEEE Transactions on Computers* **2000**, *49*, 465–481. https://doi.org/10.1109/12.859540.
29. Mário, V.; Lopes, J.D.; Véstias, M.P.; de Sousa, J.T. Implementing CNNs Using a Linear Array of Full Mesh CGRAs. In Proceedings of the Applied Reconfigurable Computing (ARC 2020), Cham, 2020; Vol. 12083, *Lecture Notes in Computer Science*, pp. 288–297. https://doi.org/10.1007/978-3-030-44534-8_22.

 doi:10.20944/preprints202607.1442.v1




30. Pestana, D.; Miranda, P.R.; Lopes, J.D.; Duarte, R.P.; Véstias, M.P.; Neto, H.C.; de Sousa, J.T. A Full Featured Configurable Accelerator for Object Detection With YOLO. *IEEE Access* **2021**, *9*, 75864–75877. https://doi.org/10.1109/ACCESS.2021.3082196.
31. Miranda, P.R.; Pestana, D.; Lopes, J.D.; Duarte, R.P.; Véstias, M.P.; Neto, H.C.; de Sousa, J.T. Configurable Hardware Core for IoT Object Detection. *Future Internet* **2021**, *13*, 280. https://doi.org/10.3390/fi13110280.
32. de Sousa, J.T.; Rodrigues, C.A.; Barreiro, N.; Fernandes, J.C. Building Reconfigurable Systems Using Open Source Components. In Proceedings of the X Jornadas sobre Sistemas Reconfiguráveis (REC 2014), 2014.
33. OpenRISC Project. OpenRISC 1000 Architecture Manual. https://openrisc.io/architecture.html, 2012.
34. OpenCores. ORPSoC: OpenRISC Reference Platform System-on-Chip. https://opencores.org/projects/orpsoc, 2012.
35. Herveille, R. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores. OpenCores.org, 2002.
36. Williams, S. Icarus Verilog. http://iverilog.icarus.com, 2002.
37. Snyder, W. Verilator. https://www.veripool.org/verilator, 2024.
38. Fiolhais, L.; de Sousa, J.T. Warpbird: an Untethered System on Chip Using RISC-V Cores and the Rocket Chip Infrastructure. In Proceedings of the XIV Jornadas sobre Sistemas Reconfiguráveis (REC 2018), 2018, pp. 42–49.
39. Asanović, K.; Avizienis, R.; Bachrach, J.; Beamer, S.; Biancolin, D.; Celio, C.; Cook, H.; Dabbelt, D.; Hauser, J.; Izraelevitz, A.; et al. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.
40. Bachrach, J.; Vo, H.; Richards, B.; Lee, Y.; Waterman, A.; Avižienis, R.; Wawrzynek, J.; Asanović, K. Chisel: Constructing Hardware in a Scala Embedded Language. In Proceedings of the Proceedings of the 49th Design Automation Conference (DAC), 2012, pp. 1216–1225. https://doi.org/10.1145/2228360.2228584.
41. Fiolhais, L.; Gonçalves, F.; Duarte, R.P.; Véstias, M.P.; de Sousa, J.T. Low Energy Heterogeneous Computing with Multiple RISC-V and CGRA Cores. In Proceedings of the 2019 IEEE International Symposium on Circuits and Systems (ISCAS). IEEE, 2019. https://doi.org/10.1109/ISCAS.2019.8702519.
42. Papon, C. VexRiscv: A FPGA-Friendly 32-bit RISC-V CPU Implementation. https://github.com/SpinalHDL/VexRiscv, 2017. GitHub repository, accessed June 2026.
43. Schiavone, P.D.; Rossi, D.; Pullini, A.; Di Mauro, A.; Gürkaynak, F.; Benini, L. Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V Cores for Internet-of-Things Applications. In Proceedings of the Proceedings of the 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS), 2017. https://doi.org/10.1109/PATMOS.2017.8106976.
44. IObundle. Versat-AI. https://github.com/IObundle/versat-ai, 2026.
45. Moreano, N.; Borin, E.; de Souza, C.; Araujo, G. Efficient datapath merging for partially reconfigurable architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **2005**, *24*, 969–980. https://doi.org/10.1109/TCAD.2005.850844.
46. Fazlali, M.; Zakerolhosseini, A.; Gaydadjiev, G. Efficient datapath merging for the overhead reduction of run-time reconfigurable systems. *The Journal of Supercomputing* **2012**, *59*, 636–657. https://doi.org/10.1007/s11227-010-0458-3.
47. Carraghan, R.; Pardalos, P.M. A Note on Finding the Maximum Clique. *Operations Research Letters* **1990**, *9*, 375–382. https://doi.org/10.1016/0167-6377(90)90057-I.
48. Tomita, E.; Seki, T. An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique. *Discrete Applied Mathematics* **2003**, *120*, 199–207. https://doi.org/10.1016/S0166-218X(02)00295-6.
49. Prosser, P. Exact Algorithms for Maximum Clique: A Computational Study. *Algorithms* **2012**, *5*, 545–587. https://doi.org/10.3390/a5040545.
50. Moon, J.W.; Moser, L. On Cliques in Graphs. *Israel Journal of Mathematics* **1965**, *3*, 23–28. https://doi.org/10.1007/BF02760024.
51. Microsoft. ONNX Runtime. https://onnxruntime.ai, 2018.
52. Räiskilä, K. ONNX2C: Open Neural Network Exchange to C Compiler. https://github.com/kraiskil/onnx2c, 2020. GitHub repository, accessed June 2026.
53. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778. https://doi.org/10.1109/CVPR.2016.90.
54. Krizhevsky, A. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, 2009.

 doi:10.20944/preprints202607.1442.v1




55. Koizumi, Y.; Saito, S.; Uematsu, H.; Harada, N.; Imoto, K. ToyADMOS: A dataset of miniature-machine operating sounds for anomalous sound detection. In Proceedings of the 2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA). IEEE, 2019, pp. 313–317.
56. Zhang, Y.; Suda, N.; Lai, L.; Chandra, V. Hello Edge: Keyword Spotting on Microcontrollers, 2018, [1711.07128].
57. Warden, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition, 2018, [1804.03209].
58. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, 2017, [1704.04861].
59. Chowdhery, A.; Warden, P.; Shlens, J.; Howard, A.; Rhodes, R. Visual wake words dataset. *arXiv preprint arXiv:1906.05721* **2019**.